In the Specification:

Please amend the paragraph beginning on page 6, line 4 as follows:

The preferred embodiment makes use of certain aspects of ~~UNIX-like~~ UNIX®-like operating systems to implement global breakpoints in a less intrusive fashion. In the following description, a page refers to a unit of main memory. Many modern processor architectures provide facilities to, and many modern operating systems do, deal with the main memory in fixed size units called pages.

Please amend the paragraph beginning on page 6, line 10 as follows:

The method for inserting global breakpoints includes the following steps:

1. To handle breakpoints in pages that are to be loaded into memory in the future, registering a routine to be called when pages for that code module are loaded into physical memory (inode->readpage function in ~~Linux~~ LINUX® for example). This routine inserts all the breakpoints for that page by placing breakpoint instructions at specified locations. The debugger must be able to determine which breakpoints are located on a particular page in memory. This information is generally available, though the exact way by which the information is obtained depends on the method used to specify global breakpoints.
2. Locating pages of the code module that are already present in memory, and inserting the breakpoints in the pages.
3. Detecting any private copies of pages that may have been created in different process contexts that use this module and inserting the breakpoints in those pages.

Once inserted, the breakpoints remain there since private pages of executable files are not discarded but swapped to a swap device.

Please amend the underlined text on page 6, line 30 as follows:

~~Linux~~ LINUX® Specific Implementation Details (Insertion)

Please amend the paragraph beginning on page 7, line 31 as follows:

The executable code segment in memory maps, and is backed by, the executable program image on the hard disk 134. The executable program image file 102 loaded into memory is represented by an operating system data structure, known as the inode in the ~~Linux~~ LINUX® operating system. This inode structure defines the functions that can be used to perform various file-related operations on the inode. One of the operations defined by the inode structure is the readpage function. This function is used to read the contents of the file from hard disk 134 into memory. The debugging tool 120 replaces the original readpage function 130 of the inode with its own function 124. Whenever the page fault handler 112 determines that a page of the executable file needs to read into memory, operating in conjunction with the memory manager 114, the page fault handler 112 calls the readpage function of the inode, which is now 124. This readpage function 124 first calls the original readpage function 130 to actually read the page into memory with the assistance of relevant file systems and disk device drivers denoted by the block 132.

Please amend the paragraph beginning on page 9, line 29 as follows:

A code page may possibly be 'dirtied' due to a breakpoint already inserted into the code page earlier. While what happens in such cases is operating system specific, many modern operating systems (OS), including ~~Linux~~ LINUX®, handle writing to the code segment by making private-per-process code pages. This is known as the Copy-On-Write (COW) mechanism, the consequence of which is that the page is marked as dirty. Once written to, the code pages do not correspond to the executable image file on the hard disk any longer. If these code pages ever need to be temporarily removed from the main memory due to the memory pressures, these pages are saved on the hard disk (also called the swap device). This process of temporarily removing parts of main memory and saving the parts to hard disk for later retrieval, to make more space available in the main memory, is called swapping. There is no need to intercept the readpage function for this page, since the changes made are written back, saved, in the swap device.

Please amend the paragraph beginning on page 10, line 29 as follows:

The loop subprocess 220 is carried out for each of the global breakpoints specified. In block 222, using the facilities provided by the operating system (find_page function in ~~Linux~~ LINUX® kernel), the memory page corresponding to a given inode and offset is looked up. The page determined by the lookup function 222 is provided to decision block 224.

Please amend the underlined text on page 12, line 1 as follows:

~~Linux~~ LINUX® Specific Implementation Details (Removal

Please amend the paragraph beginning on page 12, line 13 as follows:

The method of identifying, inserting, and removing global breakpoints as detailed above has the following advantages:

1. Minimally intrusive: the only hook required is in the routine that loads a page of code from the executable image. The presence of inode_operations in many ~~UNIX-like~~ UNIX®-like Operating Systems allows this to be done optimally, ensuring that there is no additional overhead when loading pages from executables with no breakpoints inserted.
2. On platforms like ~~Linux~~ LINUX® where many page faults can occur, mainly to create per-process page table mappings to pages already present in memory, this approach is advantageous because spurious page faults do not arise. This method allows the debugger to intervene and insert breakpoints at the only place where they are actually necessary, i.e. when the pages are read into memory.
3. The problem of reinserting breakpoints when discarded code pages are brought back into memory is seamlessly handled. The fact that the same inode_operation is carried out when reloading discarded code pages makes this possible.
4. All the code pages with breakpoints are not required to be present in memory when inserting breakpoints, nor is this caused to happen, ensuring that programatically inserting a large number of breakpoints in one module does not cause any significant overhead.
5. Generally process-level (application) debuggers operate by inserting breakpoints on private-per-process code pages. The approach outlined above ensures that the global breakpoint facility can function correctly and consistently even in the

presence of other debuggers.

Please amend the paragraph beginning on page 15, line 6 as follows:

Finally, while the preferred embodiment is implemented using the ~~Linux~~ LINUX® operating system, it will be appreciated by those skilled in the art in view of this disclosure that the invention can be practiced with other ~~Unix-like~~ UNIX®-like operating systems, such as Sun, HP, and AIX.

Please amend the paragraph beginning on page 15, line 11 as follows:

The methods according to the preferred embodiment utilise characteristics of the ~~Linux~~ LINUX® operating system. However, the embodiments have application to different operating system, requiring some modification. The advantages in terms of efficiency and non-intrusiveness may not be achievable to the extent possible in the ~~Linux~~ LINUX® OS, in case the other OS does not support similar characteristics. The following is a list of significant aspects that are depended on at a conceptual level:

1. The OS provides a mechanism through which the logic that loads code/data into memory from a particular file can be hooked into. In ~~UNIX-like~~ UNIX®-like systems (e.g. SUN), where there is an evolved virtual file system interface enabling specialised file system implementations including filter file systems where the low-level file system interface routines can be overridden or intercepted at the granularity of an individual file, this should be possible.

On an operating system like Windows NT, where file system routines may not be directly interceptable at individual file level but rather at a drive level, the performance is not as good, since there is an extra check happening for all files on that file system.

Also, since the main requirement is to be able to intercept the actual loading of code pages from executable files, which are typically mapped into memory, this kind of interception has to support memory-mapped input/output (IO) situations where this loading may be triggered via a page fault. That is likely to be the case wherever support for layered file systems is intentionally built in.

2. The OS maintains sufficient information to be able to track down the physical pages that have been loaded directly from a particular executable file (i.e. physical pages backed by that file). This is likely to be the case on many operating systems, where executable code pages are common (read-shared) across all processes running that executable. For any new process that is to run the same executable, the OS needs this information to cause the necessary mappings to the existing loaded pages to occur.

3. The OS maintains sufficient information, even if via indirect means, to locate all the private-copy pages that have been generated via the copy-on-write mechanism from a given executable page. The indirect means suggested in the preferred embodiment depends on the OS maintaining a list of all the address spaces in

which portions of a particular file are mapped and a way to correlate the file offsets to the corresponding virtual addresses of the mappings. The former may not be supported in some operating systems, like Windows NT. The absence of such support or a way to get to the page table mappings in all process contexts where a given physical page (backed by a file) is mapped, makes difficult providing for invalidation of a particular physical page if needed. This has some ramifications in terms of limitations of the OS, e.g. in supporting cache coherency for memory mapped files in distributed file system client implementations.

On an OS that does not provide the required support, the embodiments of the invention functionality are limited to the extent of not being able to insert global breakpoints on existing private-copy pages for that executable. For example, if the executable is already being debugged by an application debugger that has placed a breakpoint of its own, that particular private-copy page might get missed.

The remaining assumptions relate to page table based memory management approach, and the OS maintaining virtual address range descriptors for each address space, which is common to several operating systems. Any OS that addresses these aspects can be used for implementing the embodiments of the invention.